"Express Mail" Mailing Label No. **EK483826441US**

PATENT APPLICATION
ATTORNEY DOCKET NO. SUN-P6438-RSH

METHOD AND APPARATUS FOR REMOVING CLASS INITIALIZATION BARRIERS FROM SHARED COMPILED METHODS

Inventors: Laurent P. Daynes and Grzegorz J. Czajkowski

Related Application

[0001] This application hereby claims priority under 35 U.S.C. §119 to a Provisional Patent Application entitled, "The Design of MVM—a Multitasking Virtual Machine," filed March 15, 2001 by inventors Grzegorz J. Czajkowski and Laurent P. Daynes (Application No. 60/276,409).

[0002] The subject matter of this application is related to the subject matter in co-pending non-provisional applications by the same inventors as the instant application entitled, "Method and Apparatus to Facilitate Sharing Instruction Code in a Multitasking Virtual Machine," having serial number TO BE ASSIGNED, and filing date TO BE ASSIGNED (Attorney Docket No. SUN-P6119-RSH) and "Method and Apparatus for Class Initialization Barriers and Access to Class Variables in Multitasking Virtual Machines," having serial

number TO BE ASSIGNED, and filing date TO BE ASSIGNED (Attorney Docket No. SUN-P6120-RSH).

BACKGROUND

Field of the Invention

[0003] The present invention relates to computer instruction code. More specifically, the present invention relates to a method and an apparatus for removing class initialization barriers from shared compiled methods.

Related Art

[0004] Computer programs written in languages such as JAVA™ are compiled into a platform-independent code, which is executed on a virtual machine, such as a JAVA VIRTUAL MACHINE (JVM). A program that has been compiled into a platform-independent code has the advantage that it can execute on any virtual machine that supports the platform-independent code regardless of what type of underlying central processing unit and native code are used to implement the virtual machine. The terms JAVA, JVM, and JAVA VIRTUAL MACHINE are trademarks or registered trademarks of SUN Microsystems, Inc. of Palo Alto, California.

[0005] A virtual machine typically includes an interpreter, which interprets the platform-independent code into native code to perform the desired operations. Interpreting the platform-independent code is an inherently slow operation. Therefore, many virtual machine implementations also include a dynamic compiler, which dynamically compiles the platform-independent code at runtime into the native code for the machine being used to host the virtual machine. Compiling the platform-independent code into the native code for the

host machine can reduce the execution time of the program and, therefore, increase throughput.

[0006] Virtual machines for object-oriented programming languages with dynamic class loading typically load the code of a class when a program resolves a symbolic reference to that class for the first time. The class needs to be initialized subsequently when the program uses it for the first time. Loading and initialization of a class are two separate events. Initialization of a class may never take place even though the class has been loaded before. In the case of the Java programming language, the initialization of a class includes executing the class's static initializer, which brings the class's variables (also known as the static variables) to a well-defined initialized state. A virtual machine implementation may choose to set a class to the initialized state upon its loading when no action is required to initialize that class. For instance, in the Java programming language, no action is required to initialize a class when this class has no declared static initialization sequence, and either no non-final static variables, or non-final static variables that are all declared to be set to a default value. In this case, a virtual machine implementation can benefit from setting such initialization-less classes to the initialized state upon class loading.

[0007] A class initialization barrier is a sequence of native instructions that calls the virtual machine's runtime to initialize a class if the class is not already initialized. Class initialization barriers are included in the implementation of those platform-independent instructions that may result in the very first use of a class (in the case of the Java programming language, there are four such instructions: getstatic, putstatic, invokestatic, new). The implementation of a platform-independent instruction can come in two flavors: (i) as a sequence of instructions that is part of the implementation of an interpreter of platform-

independent instructions, (ii) or as a sequence of instructions generated by a dynamic compiler from platform-independent instructions.

[0008] Dynamic compilers have two ways to eliminate class initialization barriers. Simple static analysis (complex static analyses are most of the time out of question because of their cost, which is unacceptable for a runtime compiler), and code patching.

[0009] Dynamic compilers take advantage of their knowledge about the current runtime state of an executed program to eliminate class initialization barriers at compile-time: typically, class initialization barriers are not emitted for classes that have been initialized. For those cases where a class hasn't been initialized by the time the compilation takes place, self-rewriting code is generated instead in order to eliminate the class initialization barriers at execution time.

[0010] In order to save processing and memory, a multitasking virtual machine (MVM) aims at sharing as much of the runtime representation of a class as possible between tasks. Targets for sharing include the platform-independent code, the meta-data describing the class, and the native code produced by the dynamic compiler. Sharing the latter across sequential or concurrent execution of a program offers many advantages: it factors out the costs of runtime compilations, it eliminates the need for interpretation since compiled code is immediately available, and it reduces the space overhead of compiled methods when platform-independent programs are run simultaneously.

[0011] Unfortunately, sharing compiled code invalidates current class initialization barrier removal techniques because of dynamic loading: classes may be loaded in different order by different programs, or even by multiple executions of the same program. Therefore, an assumption that is correct for one execution (e.g., class A is initialized) may be incorrect in another. Because of this, class initialization barriers can be omitted only if the order of class initialization is

guaranteed to be always the same for all possible executions of any programs using these classes.

[0012] Class initialization barrier elimination is also important to enable some optimization techniques, such as, the inlining of static methods. Inlining can bring significant performance gains, especially for methods that are frequently used. Most dynamic compilers do not attempt to inline a method if its class hasn't been initialized by the time the compilation takes place. If class initialization barriers can be eliminated for a particular static method call site, then it is possible to inline the method at this call site (provided that other inlining conditions apply as well).

[0013] What is needed is a method and an apparatus, which allows removal of class initialization barriers from shared compiled methods that do not exhibit the problems defined above.

SUMMARY

[0014] One embodiment of the present invention describes a mechanism that determines during the dynamic compilation of a method M of a class C if a class initialization barrier can be omitted in the native code produced by the compilation of M based on annotations made to classes during their initialization. The mechanism works by first annotating the shared runtime representation of classes that are initialized during the startup sequence of the very first task executed by the multitasking virtual machine with information that identify the particular event that triggered the initialization of these classes, and in particular, if that event is a class initialization barrier from a method of another class. Classes initialized during the startup of the first task executed by the multitasking virtual machine are called "bootstrap" classes. Irrespective of the task executed, bootstrap classes are always initialized in the same order, and their initialization is

always triggered by the same events. Classes that are initialized after the startup sequence of the first task executed by the multitasking virtual machine have a "blank" annotation. The annotation of a class already annotated is left unchanged by subsequent initialization of the class by other tasks (in a multitasking virtual machine, a class can be initialized up to one time per-task executed by the virtual machine). Annotations are then used during runtime compilation of a method M of a class C by any tasks to determine if a class initialization barrier site can cause the initialization of class D targeted by the class initialization barrier. If D's annotation is blank, the mechanism cannot tell whether the class initialization barrier can be omitted, and the dynamic compiler must rely on other means to decide if it can omit the barrier. If D's annotation indicates that D is a bootstrap class, native code for the class initialization barrier is emitted only if D's annotation further indicates that the class initialization barrier site may have triggered D's initialization.

[0015] In one embodiment of the present invention, a class is annotated when its initialization state is atomically set to the fully initialized state. When calling the multitasking virtual machine runtime to initialize a class D, a class initialization barrier passes information that identifies, more or less exactly, the location of the initialization barrier. This information is carried by the multitasking virtual machine until class D becomes fully initialized, at which point the information is used to annotate class D. If initialization of class D is triggered by a event different than a class initialization barrier, this information is carried by the multitasking virtual machine until class D fully initialized, at which point the information is used to annotate class D.

[0016] In one embodiment of the present invention, the annotation to a class C is a pointer to the shared runtime representation of a class. A NULL pointer value indicates a blank annotation. A pointer to the shared runtime

representation of any class indicates that C is a bootstrap class. A pointer to the shared runtime representation of a class D, such that D is different from C, indicates that a class initialization barrier of any method of D may have triggered C's initialization. A pointer to the shared runtime representation of class C itself indicates that C's initialization wasn't triggered by a class initialization barrier.

[0017] In one embodiment of the present invention, the annotation to a class C is a pointer to the shared representation of a method M. A NULL pointer value indicates a blank annotation. A pointer to the shared runtime representation of any method indicates that C is a bootstrap class. A pointer to the shared runtime representation of a method M of a class D, such that D is different from C, indicates that a class initialization barrier of method M of D may have triggered C's initialization. A pointer to a non-NULL value chosen so that the value cannot represent a pointer to the shared runtime representation of any method indicates that a class initialization barrier did not trigger C's initialization.

[0018] In one embodiment of the present invention, a dynamic compilation of a method M of a class C does not generate the native code of a class initialization barrier that targets a class D if C and D are the same. The barrier is unnecessary in this case because in order to execute M, C (and, therefore D) must be already initialized.

[0019] In one embodiment of the present invention, a dynamic compilation of a method M of a class C does not generate the native code of a class initialization barrier that targets a class D if C inherits from D. The barrier is unnecessary in this case because in order to execute M, C must be already initialized, and the class C inherits directly from must be initialized before C is initialized.

[0020] In one embodiment of the present invention, a dynamic compilation of a method M of a class C does not generate the native code of a

class initialization barrier that targets a class D if D is initialization-less (see [0006]).

BRIEF DESCRIPTION OF THE FIGURES

[0021] FIG. 1 illustrates computing device 100 including multitasking virtual machine 102 in accordance with an embodiment of the present invention.

[0022] FIG. 2 illustrates multitasking virtual machine 102 in accordance with an embodiment of the present invention.

[0023] FIG. 3 is a flowchart illustrating the process of determining whether native code for a class initialization barrier should be generated by the dynamic compiler in accordance with an embodiment of the present invention.

[0024] FIG. 4 is a flowchart illustrating the process of determining whether native code for a class initialization barrier should be generated by the dynamic compiler in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

[0025] The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art, and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

[0026] The data structures and code described in this detailed description are typically stored on a computer readable storage medium, which may be any

device or medium that can store code and/or data for use by a computer system. This includes, but is not limited to, magnetic and optical storage devices such as disk drives, magnetic tape, CDs (compact discs) and DVDs (digital versatile discs or digital video discs), and computer instruction signals embodied in a transmission medium (with or without a carrier wave upon which the signals are modulated). For example, the transmission medium may include a communications network, such as the Internet.

Overview

[0027] The present invention contains a mechanism to annotate classes with information that help identifying the event that triggered their initialization, and a simple set of tests that use this information, among others, to establish when the generation of a class initialization barrier can be omitted when compiling a method of a class into native code that can be shared between multiple tasks of a multitasking virtual machine. One advantage of avoiding the generation of a class barrier initialization is allowing method inlining. The information classes are annotated with take advantage of the initialization order of classes that are initialized during the bootstrap sequence of a task of the multitasking virtual machine, called hereafter bootstrap classes. The following description details some background on bootstrap classes, how and when they are initialized, and how classes are represented in a multitasking virtual machine, before detailing the annotation mechanism and how it is used to determine if code must be generated for a class initialization barrier at compile-time.

[0028] A virtual machine always performs a bootstrap sequence that brings the virtual machine to a state where it can start executing the instructions of a platform-independent program, which it has been instructed to execute.

[0029] A multitasking virtual machine typically uses two bootstrap methods: a heavyweight one that is used when the multitasking virtual machine is launched to execute its first program; and a lightweight bootstrap sequence for any subsequent programs the virtual machine is instructed to execute. The heavyweight bootstrap sequence is analogous to the bootstrap sequence of standard (i.e., non multitasked) virtual machines. The lightweight bootstrap sequence is semantically equivalent to the heavyweight one, but much faster. It establishes an execution context that is specific to the new program and that isolates it from any interference with other programs that the multitasking virtual machine may be executing simultaneously.

[0030] In both cases, the bootstrap sequence requires initializing a mandatory set of platform-independent classes essential to the execution of the virtual machine itself. These classes are known as the "bootstrap" classes and are a subset of what is commonly called the "core" classes. Initializing a class consists of executing a, potentially empty, sequence of instructions that brings the class, and in particular, its variables, to an initial state where the class can be used by a program. In the case of the Java programming language, the initialization of a class consists of executing the static initializers and the initializers for static variables declared in the class.

[0031] The order in which the bootstrap classes are initialized is fixed and always the same for every execution of any platform-independent programs (for a given version of the virtual machine and of the core classes). The present invention takes advantage of this fact to eliminate class initialization barriers directed at bootstrap classes at compile-time. Not all class initialization barriers to bootstrap classes can be eliminated though, since a multitasking virtual machine, in contrast to a non-multitasking virtual machine, may execute a compiled method at bootstrap.

[0032] A multitasking implementation of the virtual machine typically splits the runtime representation of a class into two sets of objects: one that can be shared by all programs it executes, and one that is private to each program, and replicated for each program. In the case of a virtual machine for the JAVA programming language, bytecodes, constant pools, methods, fields and exception tables are examples of what is shared between programs, whereas, static variables, class initialization states, and class loader objects are examples of what is replicated for each program. The construction of the shared part of the runtime representation of a class is done the first time the class is loaded by a program. In particular, the shared part of the representation of a bootstrap class is built only when the multitasking virtual machine is started.

[0033] By definition, the bootstrap sequence of all tasks, and in particular, of the first task executed by a multitasking virtual machine, is a sequence of operations performed serially (i.e., non-concurrently) by the virtual machine before the task is ready to execute the program it was instructed to run, and before the task starts the execution of new threads of control that may create concurrency. In other words, the bootstrap sequence of a task is concurrency-free. Three properties of a multitasking virtual machine make both the order in which bootstrap classes are initialized, and the event that triggers the initialization of each bootstrap class, task-independent. These properties are:

- there is no concurrency during the bootstrap of a task executed by the multitasking virtual machine;
- the effect of a task are strictly isolated from other tasks; and,
- all programs executed by the multitasking virtual machine are forced to use the same binary copy of bootstrap classes.

[0034] The present invention augments the shared runtime representation of each class C with a reference to an object called the initializer of C, which

contains information that identify, more or less precisely, the event that triggered the initialization of C. However, the dynamic compiler cannot rely on the initializer of non-bootstrap classes to decide the elimination of a class initialization barrier since the initializer of such classes is not task-independent. For this reason, the annotation mechanisms leave the initializer of any classes initialized outside of the bootstrap sequence blank, which translates into setting the initializer of such classes to the NULL pointer value.

[0035] In all cases, the value of the initializer of a class is constant across all tasks: either it is a bootstrap class, in which case the class is always triggered by the same event and the initializer of the class always point to the same object that represents that event; or it is not a bootstrap class, and the value of its initializer must always be NULL. For this reason, the initializer of a class needs to be set only once, upon the first time the class is initialized. One way to achieve this is to maintain in a global variable of the multitasking virtual machine a Boolean value that indicates whether the bootstrap sequence of the first task is completed. Class initialization consults this variable to decide whether to assign the initializer of a class or not. The following illustrates a function that may be used to set the initializer of the class when the class is atomically set to the fully initialized state:

```
        set_initializer(SharedRuntimeClassRepresentation
C, Object X){
            if (in_first_bootstrap &&
initializer(C)==NULL){
                initializer(C)=X;
            }
        }
```

The code above assumes that, upon creation of a shared runtime class representation, its initializer field is set to the NULL value.

[0036] The value of the initializer of a class is specified by the event that triggers the initialization of the class, and carried by the multitasking virtual machine until the class is atomically set to the fully initialized state. Class initialization is triggered either from a class initialization barrier of a method that is being executed, or directly by the virtual machine runtime. Examples of the latter case include initializing the super class of a class that is being initialized, or initializing a class because the virtual machine runtime needs the class to be initialized at a well-defined time.

[0037] The invention allows two types of value for the initializer of a bootstrap class: a pointer to the shared runtime representation of a class, or a pointer to the shared runtime representation of a method. In both cases however, the initializer of a non-bootstrap class is always set to a NULL pointer value, and the initializer of a bootstrap class C whose initialization is not triggered by a class initialization barrier is set the pointer to the shared representation of C itself. In the latter case, any other encoding is allowed as long as it is guaranteed to be distinguishable from the encoding of any initializer of a bootstrap class whose initialization is triggered by a class initialization barrier. The following describes in detail the two favored ways to encode the initializer of a class, and how, in each case, this information is used by a dynamic compiler to decide on generation of the native code for a class initialization barrier when compiling a method.

[0038] A multitasking virtual machine may assign the pointer to the shared runtime representation of class D to the initializer of a bootstrap class C when a class initialization barrier of a method of D triggers the initialization of C.

[0039] Given this, a dynamic compiler can use the simple following sequence of tests to determine if a class initialization barrier can be omitted from

the compiled code of a method, so that multiple programs can share this compiled code. C is the class of the method M being compiled, and X is the class being targeted by a platform-independent instruction that may trigger the initialization of X. The dynamic compiler can omit the generation of a class initialization barrier for such instructions if the following function returns false:

```
boolean needs_CIB(X, C) {
  if (X == C) {                              // Case 1
    return false;
  } else if (X is superclass of C){          // Case 2
     return false;
  } else if ( (initializer(X)!=NULL) && (initializer(X)
!= C)){                                      // Case 3
    return false;
  }
  return true;                               // Case 4
}
```

Case 1. X and C are the same class. Since methods can only be invoked if the class that defined them is already initialized, barriers that target the class that defines the method that includes them are unnecessary. In other words, since m is a method of X, class initialization barriers that apply to X within m are unnecessary.

Case 2. C inherits from X (i.e., X is a super-class of C). The initialization of a class requires the initialization of all of its superclasses beforehand (since they participate in the definition of the class). Since m can only be executed once C is initialized (see comment on case 1), it follows than m

can only be executed if C's superclass, and in particular X, are already initialized. Hence, barriers that target superclasses of the class that defines the method that includes them are unnecessary. In other words, since m is a method of subclass of X, class initialization barriers that apply to X within m are unnecessary.

Case 3. (initializer(X) != NULL) && (initializer(X)!=C)
X's initializer is not NULL, which means that X is a bootstrap class. In that case, a class initialization barrier is needed only if C is the initializer of X, since the method M may be the method that triggered X's initialization.

Note that this test encompasses the case where initializer(X) == X: since X!=C because of case 1, initializer(X)==X implies that the (initializer(X)!=NULL && initializer(X)!=C) is true. Therefore, when the value of initializer(X) is X itself, the function needs_CIB returns false and no native code will be generated by the dynamic compiler for this barrier. This is the desired effect when X is a bootstrap class that is not initialized by a class initialization barrier (as indicated by initializer(X)==X), since, in that case, no method can ever trigger the initialization of X.

Case 4. In the absence of any other information, the runtime compiler must be conservative and must emit the native code for a class initialization barrier.

[0040] A multitasking virtual machine may assign the pointer to the shared runtime representation of method m to the initializer of a bootstrap class C when a class initialization barrier of method M triggers the initialization of C.

Using methods instead of classes for annotating bootstrap classes enables to gain in precision, and potentially decrease the number of class initialization generated by the runtime compiler.

[0041] The method needs_CIB is changed as follows to reflect this more precise variation:

```
boolean needs_CIB(X,M){
   if (X == M.method_holder()) {    // case 1
     return false;
   } else if ( X is superclass M.method_holder()){
                                    // case 2
     return false;
   } else if (initializer(X)!=NULL &&
initializer(X)!=M){                 // case 3
     return false;
   }
   return true;                     // case 4
}
```

[0042] Under this approach, M is the method being compiled, M.method_holder() denotes the class that defines method M, and X is the class being targeted by a class initialization barrier of M.

[0043] As said earlier, the multitasking virtual machine may take advantage of initialization-less classes to avoid generating class initialization barrier for instruction that applies to such class, irrespectively of whether the class is a bootstrap class or not. In this case, a multitasking virtual machine set the initialization state of the initialization-less class to fully initialized upon its loading. In order to avoid generating class initialization barrier code against an

initialization-less class C in compiled methods, it is enough to set the value of C's initializer to C itself. That way, C will always fall in case 3 of each of the needs_CIB functions described above. This technique works irrespectively of the type of initializer information chosen for bootstrap class (i.e., a pointer to a shared runtime representation of a class, or a pointer to a shared runtime representation of a method).

Computing Device

[0044] FIG. 1 illustrates computing device 100 including multitasking virtual machine 102 in accordance with an embodiment of the present invention. Computing device 100 can generally include any type of computer system, including, but not limited to, a computer system based on a microprocessor, a mainframe computer, a digital signal processor, a portable computing device, a personal organizer, a device controller, and a computational engine within an appliance.

[0045] Multitasking virtual machine 102 is a virtual machine, which has multitasking capability. As shown if FIG. 1, multitasking virtual machine 102 includes shared runtime system 112, tasks 106, 108, and 110, and dynamic compiler 114. Tasks 106, 108, and 110 are representative of a task load. A practitioner with ordinary skill in the art will readily recognize that multitasking virtual machine 102 can include any number of tasks, depending only on the resources available within computing device 100. Dynamic compiler 114 compiles platform-independent codes into the native code for computing device 100.

Multitasking Virtual Machine

[0046] FIG. 2 illustrates multitasking virtual machine 102 in accordance with an embodiment of the present invention. Multitasking virtual machine 102 includes shared runtime system 112, tasks 106, 108, and 110, and dynamic compiler 114 from FIG. 1. Shared runtime system 112 includes shared representations of various classes including shared representation of class A 202. Shared runtime system 112 also includes initializer 210. Tasks 106, 108, and 110 include task-private representation of class A 204, 206, and 208, respectively.

[0047] Shared representation of class A 202 includes constant pools, methods, fields, exception tables, and the like, which are shared by the tasks executing within multitasking virtual machine 102. Task private representation of class A 204, 206, and 208 include static variables, class initialization states, class loader objects, and the like that are private to tasks 106, 108, and 110, respectively.

[0048] Initializer 210 is a variable associated with shared representation of class A 202. The function, and possible values, of initializer 210 was covered in detail above in the overview of the present invention.

[0049] Dynamic compiler 114 compiles platform-independent codes into the native code for computing device 100. In operation, multitasking virtual machine 102 determines when a method of a class is being used sufficiently often so that the cost of compiling the platform-independent code into native code for computing device 100 can be recovered by the increased performance of the native code.

Removing Class Initialization Barriers From Compiled Code

[0050] FIG. 3 is a flowchart illustrating the process of determining whether native code for a class initialization barrier should be generated by the

dynamic compiler in accordance with an embodiment of the present invention. The system starts when dynamic compiler 114 processes a platform-independent instruction from the platform-independent code stream for a method M of a class C, wherein the instruction may trigger the initialization of a class X. In the JAVA programming system there are four such instructions: getstatic, putstatic, new, and invokestatic.

[0051] Next, dynamic compiler 114 determines if class X and class C are the same (step 306). If so, the process proceeds to step 318, otherwise the process proceeds to step 308.

[0052] At step 308, dynamic compiler 114 determines if class X is a superclass of class C (step 308). If so, the process proceeds to step 318, otherwise the process proceeds to step 310.

[0053] At step 310, dynamic compiler 114 examines initializer 210 of class X (step 310). Next, dynamic compiler determines if initializer 210 is NULL (step 312). If so, the process proceeds to step 316, otherwise the process proceeds to step 314.

[0054] At step 314, dynamic compiler 114 determines if initializer 210 is equal to class C (step 314). If so, the process proceeds to step 316, otherwise the process proceeds to step 318.

[0055] At step 316, dynamic compiler 114 is instructed to generate native code for a class initialization barrier targeted at class X. At step 318, dynamic compiler 114 is instructed to skip the generation of native code for a class initialization barrier.

Alternate Method of Removing Class Initialization Barriers

[0056] FIG. 4 is a flowchart illustrating the process of determining whether native code for a class initialization barrier should be generated by the

dynamic compiler in accordance with an embodiment of the present invention. The system starts when dynamic compiler 114 processes a platform-independent instruction from the platform-independent code stream for a method M of a class C, wherein the instruction may trigger the initialization of a class X. In the JAVA programming system there are four such instructions: getstatic, putstatic, new, and invoke static.

[0057] Next, dynamic compiler 114 determines if class X and class M.H, the class that defines method M, are the same (step 406). If so, the process proceeds to step 418, otherwise the process proceeds to step 408.

[0058] At step 408, dynamic compiler 114 determines if class X is a superclass of class M.H, the class that defines method M (step 408). If so, the process proceeds to step 418, otherwise the process proceeds to step 410.

[0059] At step 410, dynamic compiler 114 examines initializer 210 of class X (step 410). Next, dynamic compiler 114 determines if initializer 210 is NULL (step 412). If so, the process proceeds to step 416, otherwise the process proceeds to step 414.

[0060] At step 414, dynamic compiler 114 determines if initializer 210 is equal to method M (step 414). If so, the process proceeds to step 416, otherwise the process proceeds to step 418.

[0061] At step 416, dynamic compiler 114 is instructed to generate native code for a class initialization barrier targeted at class X. At step 418, dynamic compiler 114 is instructed to skip the generation of native code for a class initialization barrier.

[0062] The foregoing descriptions of embodiments of the present invention have been presented for purposes of illustration and description only. They are not intended to be exhaustive or to limit the present invention to the forms disclosed. Accordingly, many modifications and variations will be apparent

to practitioners skilled in the art. Additionally, the above disclosure is not intended to limit the present invention. The scope of the present invention is defined by the appended claims.